Microsoft Software Coding and Design Principles V 1.0

Gordon Letwin
Ross Garmoe

Microsoft Corporation
September 24, 1984

ABSTRACT

This document describes the coding and design standards for Microsoft software development. These standards are designed to increase productivity and the sharing of code by maximizing clarity and minimizing unexpected interactions.

1. Introduction

1.1. Goals

This document describes and discusses Microsoft's internal coding and design standards. The goal of these standards is to increase productivity in two ways: first, by improving code quality and productivity, and second, by facilitating the sharing of code between projects and development teams.

The intent of this document is not to present a list of arbitrary rules to be followed by rote. Instead, we discuss the *goals* of good design and coding techniques and then present design policies which systematically obtain those goals. The thrust must always be toward the 'spirit of the law' rather than the 'letter of the law': on the rare occasions where the coding spec rules interfere with the goals of this document then those rules must be broken.

A complete and running new program is only about one third of the way through its life cycle. Another third consists of the periodic maintenance, alterations, and enhancements that will be made over the years.

The remaining third is the the "cannibalization" of the program: extracting its techniques, algorithms, structures and ideally, the code itself into other programs. When a programmer has access to a body of clean, well designed and understandable code, it is common practice to take some existing code, strip out what you don't want, and use what is left as a starting point for some new code.

When we develop code we must consider 'making it work' to be only one of our goals: we must expend equal effort towards making a program understandable and maintainable.

If we are to maintain our lead in the software industry we must treat the writing of code as the *development of technology.* When we write code to perform a particular function, we must not view it as a special item that will be used this one way in this one product. Instead we should view the code as an advancement of our software technology base. When the code is complete, we will have in our company "tool box" the technology to solve that particular problem. A great variety of future programs may draw upon that technology. Some may use it unchanged, others may use it as a starting point for developing more advanced capabilities.

Naturally we must focus on the immediate need and not over-invest effort in expanding the routine solely in anticipation of future need, but we should always view new code as an advancement of our company-wide capabilities. Code should be written in anticipation of its future reuse and enhancement, not with the view that each project starts from scratch with code re-written from ground zero.

1.2. Increasing "Understandability"

A key goal of these standards is to improve program clarity. Programs coded in a high level language contain relatively few bugs that are caused by simple 'typos'. Some bugs represent serious design omissions so that the programmer is producing code to correctly do the wrong thing, but most bugs are in the oversight category: omitted, duplicate, conflicting or incorrect code. Because of this, we say that most program bugs are the result of not understanding the code. Undoubtedly the programmer debugging his latest opus magnum would disagree - he just got through writing the code and would say that he knows it very well indeed. It is true that he knows quite a bit about the code but if he truly understood what he had written he would have no oversight-related bugs. The compiler finds most typos; after the programmer finds the rest a fully understood program will run.

As an example, the 8086 *C* optimizer had a bug that caused it to loop when optimizing certain programs. The bug was traced to a subroutine that scanned a

symbol table made up of 20 byte entries. Each entry contained a string terminated by a null character. The error arose when a string of 20 bytes was entered in the table so that the terminating null character made a total of 21 bytes and damaged the next entry. In reality, the table could only hold symbols with a maximum length of 19 characters; this was not documented. This meant that in turn, the routine which makes symbol table entry could only accept strings up to 19 characters long; this also was not documented. We could say that the symbol table routines were not understood by the maintainer who wrote the calling code. If he had understood the restrictions he would not have tried to enter strings that could possibly (and did!) exceed the limit. In a certain sense, the restriction was not understood even by the original author: if he had understood the limitations the code would have been designed to guarantee that the limitations were enforced. Either the symbol table manipulation routines needed to check their arguments or their callers needed to guarantee the size of the strings they inserted. It is not sufficient for the designer just to mentally address this issue; later some programmer will come along and change the code in one of the callers without understanding the limitation or how that limitation is enforced.

When we say the author didn't understand the code we don't mean that he was ignorant of it. In all likelihood the author could tell you that the maximum size entry is 19 characters. The problem is that a fact stored away in the back of your mind is like a lost item: yes, you have it, but no, it doesn't do you any good. A routine is *understood* only when the programmer is fully aware of its functions, limitations, and side effects. This awareness must "accompany" the routine in the programmer's head.

This example emphasizes a crucial point: proper design and documentation aid both the program maintainer and the program designer. A human being can only keep so many items in "working storage" at one time. If the programmer fails to design and document the full function, limitations and side effects of his routines, he *will* slip up somewhere.

1.3. Side Effects

The second key goal of these standards is to minimize unexpected interactions within a program. Such interactions are often the result of poor locality but can also arise when code depends upon undocumented relationships among procedures or procedures that cause undocumented side effects. These unexpected (undocumented) interactions are the source of most product stability problems - a change made to a program causing apparently unrelated problems to appear. Even worse; usually the 'broken' section of the program was not re-tested because it was thought to be unaffected by the change.

1.4. Lexical Conventions

Finally, this document will discuss stylistic and lexical conventions. A standard coding style is presented which allows one to quickly eye-scan programs without having to adjust to presentational differences. In addition, if a program is written in the standard style, deviations from the style will indicate potential errors and problem areas.

2. Abstraction

Abstraction is the keystone of successful software development. The number of information items a human brain can simultaneously process is quite limited. Some form of abstraction is required so that only a subset of items need be considered at one time. These subsets are then considered 'known' and their workings are "hidden away in a drawer". The new simplified item sets are then themselves combined. Mathematics is a highly developed example of this process. Once a theorem is proven by means of mathematical base objects the theorem itself becomes a base object for further use. The mathematician may or may not understand the proofs of the theorems he uses, but in either case he need not keep such an understanding uppermost in his thoughts.

Abstraction is well known in the field of programming, as well. It is the central concept of high-level languages and structured programming. We belabor it here to make a specific point: since the inner workings of each subroutine are 'forgotten' in the abstraction process, their workings must be completely trustworthy. By this we mean that the workings of a subroutine must do exactly what they are supposed to do, no more, no less. Any errors or uncertainties in a subroutine are compounded as the abstraction process continues. Consider the difficulties in mathematics if, by some fundamental rule of the universe, any mathematical proof had a 5% uncertainty factor: any non-trivial derivation would be worthless.

Likewise, any non-trivial program that was coded loosely is worthless, or at least, ridden with bugs. The traditional approach to programming has been to hack out some code; then test out the bugs. We are all aware of the enormous amounts of time this process requires; and since this type of testing fixes only the common and expected paths through the code the result is a bug-prone product. Proper use of abstraction not only increases productivity but as a side effect produces programs which are highly stable and comparitively error-free.

Those who have watched the rise of the Japanese in high-tech engineering should recognize the moral: it is better to build it correctly than to slap it together and then try to test out the bugs. It does require a considerable investment to learn how to build things right, but once that investment is made, the pay back in reliability and productivity is overwhelming. Of course, a software package is not a memory chip - no program can be produced truly bug-free - but proper design and testing methodologies can reduce debug time and improve product stability by orders of magnitude.

2.1. Subroutines

The fundamental unit of programming abstraction is the subroutine. The purpose of a subroutine is to combine a group of items into a smaller item which will in turn be abstracted and combined. A subroutine must *encapsulate* its contents into as small a item as possible to facilitate further levels of abstraction. This means that one should accurately describe the input, operation and output of a subroutine as concisely as possible and further, *design* subroutines to allow optimally concise description. By 'concise' we refer to a conceptual simplicity rather than a paucity of words.

A prime requirement for a subroutine is completeness of function or 'generality.' A routine should perform some specific function, only that function, and all of that function. For example, a procedure to copy a block of memory might be called:

```
/***    copyb - copy a block of bytes
 *
 *      copyb (from, to, cnt)
 *      etc...
```

This subroutine should be coded to handle overlapping fields by moving from the

front or back as necessary to avoid rippling. Further the subroutine must be coded to handle the case where 'cnt' is 0 - it promises to copy 'cnt' bytes and 0 is a perfectly good count. It is erroneous to say, "Well, I know that I'll never ask it to copy zero bytes." Some management routine might somewhere compute a zero byte count and call 'copyb' with it. If not that, later some programmer will write such a routine not knowing that this hidden restriction exists. If for some unlikely reason it is critical that 'copyb' omit the test, it should be called 'copybnz' or some other name that emphasizes the restrictions. Of course, the restriction must be fully documented in the subroutine header.

As a further example, consider a subroutine to insert items in a table. It should handle all aspects of the task - if the table overflows, the subroutine or the routines it calls should handle the allocation of more memory, as well as all other boundary conditions. If there are different insertion rules to follow under different circumstances, the subroutine should make those decisions. If this is not possible, it is better to pass a 'rule indicator' argument to a single subroutine than to have multiple routines performing similar tasks. Even if the individual rules are complex enough to require separate subroutines, only the master table routine should call them so that the job of table insertion is fully encapsulated.

Of course, it is easy to construct cases where building one master routine generates a complex and tricky architecture when partitioning the task produces a good design. Once again, the emphasis is not on a magic rule, but upon the principle of building routines whose function can be quickly and accurately conveyed without special "if's", "and's", or "but's". Usually, this is best accomplished by having one master routine handle the entire situation, calling lower level routines to handle individual cases if necessary.

Proper locality of reference and effect is essential for good abstraction. If a subroutine references a number of global variables or has an effect on the global environment of the program it is difficult for anybody (including the program designer) to develop an effective abstraction of the subroutine. In such cases the header documentation invariably refers to what the routine is INTENDED to do, not what it DOES do.

To properly document the subroutine, all global references and effects must be discussed in the subroutine header. The resultant complexity of this discussion reduces the level of abstraction for the subroutine because more information must be remembered. Therefore, effective abstraction of a subroutine or group of subroutines requires that global references and effects be minimized.

All non-local effects of a subroutine must be mentioned in its header, not just global variable references. Thus, allocating or deallocating some object is a global effect, as is modifying the interrupt level, etc. In assembly language, the modification of a register's contents is a global effect. Programming some peripheral chip or device is a global effect. In general, any action which is or might be visible outside the routine is a global effect.

3. Internal Documentation

3.1. Header Contents

The fundamental requirements of internal documentation are the same no matter what programming language is being used. The lexical details are language-dependent and are discussed later.

The main routine and every subroutine in a program should be preceded by a documentation header that describes the routine, its usage, inputs and outputs, and its side effects and restrictions. This documentation header should contain sufficient detail to allow another programmer to fully understand the function of the routine without detailed reading of the routine body.

3.2. Assumptions

This paper has discussed the importance of coding subroutines to handle all possible cases. Often, however, good design and efficiency may dictate that some reasonable assumptions be made. For example, we might write a bubble sort rather than a shell sort because we know that the routine will never be passed more than a handful of items to sort. Another possibility might be coding a routine which sets an interlock, and does not include code to handle the (perhaps unhandlable) situation where the interlock is already set. When these cases come up it is critical to mentally 'prove' that the unhandled circumstances cannot arise. Then this proof must be described in the documentation, not only in this routine's header but in the headers of routines whose characteristics were used as proof corollaries. Thus if we know that routine **findsym** will shuffle the symbol table if it must add a symbol - yet we want to call it while maintaining a pointer to a symbol table entry - we tell ourselves: "This is OK, because the caller only invokes **findsym** on symbols which we know are already in the symbol table." We then document this proof by adding a few lines to 'findsym':

```
*       This routine changes the logical addresses of
*       symbol table entries if an insertion is made.
```

We then add to the caller:

```
*       Note that we maintain 'sptr' pointing to the
*       symtab entry.  'findsym' only moves entries if
*       a symbol is added, and we know this symbol
*       exists, because we were called to assign a
*       new value to it...
```

This notation should appear in the code or the documentation header depending upon the style and complexity of the routine.

The purpose of such "proofs" is two-fold. First, we demonstrate to ourselves that the assumption is valid. Often as we go through the exercise we find that this situation can, indeed, come up. Second, by minimizing the number of "magic interdependencies" and fully documenting the remainder in all the participating modules we minimize the single biggest cause of maintenance bugs and instability: unexpected (or forgotten) side effects.

A helpful documentation technique is to continually envision a future maintenance programmer and try to anticipate the problems and mistakes that might be encountered. As you write each routine, think:

> *How might this be screwed up? If this item is changed, is it apparent that it will affect this other thing? What coding and documentation can I write, and where should I put it so that it can't be missed if this area is changed?*

Don't dwell just on the changes that you anticipate - the program or this routine might one day be adapted to some considerably different purpose. Naturally you cannot anticipate all possible changes, but try to take a "visionary" view.

Try to use techniques to simplify things as much as possible; assume a certain level of diligence on the part of the maintainer - you cannot be expected to correct for carelessness - but don't assume that he is a bloodhound. For example, if a maximum symbol length of 20 is assumed throughout a program it is not necessary to explain this at length in each routine. Instead use a symbol MAXSYM - set to 20 - and discuss any special restrictions where the symbol MAXSYM is defined. Further, make sure that all modules affecting or affected by this limit make reference to MAXSYM.

Sometimes we may be maintaining a routine which - due to complexity or poor design - we do not understand sufficiently to "prove" the validity of some restriction. This can come up even in well-written programs, such as an operating system or compiler, where the data flow and control sequence are dictated by external requests. One may be quite confident - but not absolutely sure - that a certain sequence cannot occur. In such cases it is useful to code internal consistency checks. Various key points and any questionable areas should contain checks to see if these assumptions indeed hold true. If these tests are not costly in size and time, they should be left in indefinitely. Otherwise conditional compilation should be used so they can be withdrawn when the product is released. It is important to note that just because a program comes up with the correct result, it did not necessarily get that result in the correct way! For example, consider a program to predict the winner of an election: Republican or Democrat. It might run for several years before it is discovered that it is incorrect. Consistency checks help reduce such problems because they require that the program meet both external and internal specifications when it runs. A customer would certainly prefer a message:

```
"Internal error #3: Contact Microsoft"
```

rather than getting some later, incorrect diagnostic (or worse, just incorrect results!) (Naturally consistency checks are no substitute for proper testing).

4. Peer Review

As a program is being developed the designer/programmer develops a mind-set about the program and even the most obvious flaws are overlooked. Peer review is the technique of having another programmer (a "peer") look at a program with an eye toward actual or potential problems in the design or implementation of the program. The important thing to remember is that the errors discovered during a peer review are bugs that will not have to be uncovered during debugging or after the product has been shipped. Peer review is not intended to be a structured walk-through or an opportunity for one programmer to attack another. The intent is to produce the best possible product in the minimum amount of time. In fact, if the program is large or if a 'large' number of errors are discovered, a second peer may have to be called in because the first reviewer may also develop a mind-set about the program.

Long before actual peer review would be appropriate, you can use the concept to review code as it is created. Envision yourself as another programmer, a rather critical one, and look at your work from the viewpoint: "What can I find to complain about?". Anticipate a demanding reviewer and work towards leaving him frustrated by leaving him nothing to fault. Areas meriting consideration are:

- General Structure

  If someone looks for the code that performed some specific function, can it readily be found? Can major data structures be found and understood? How easily can the major algorithm be understood and how easy is it to find the code elements that make up the backbone of that algorithm? It should be possible for someone to say, "how does this product handle so-and-so?" and - starting cold - be able to answer the question in a few minutes.

- Global Data

  Can an outsider readily learn the names and uses of the various items of global data? When a programmer changes or adds code, is it possible to understand which global items are valid and can be used? More than *what* they contain, *when* do they contain it? In other words, at what times are the values valid? Can someone readily understand the effects of modifying or removing some global items? Do some global items act as state variables? Can an outsider understand the states the program might attain? Is it clear which states are attainable and which can never be attained, and why?

- Program Structure

  Can a maintainer quickly learn the major facilities (functions) available for use? Is it possible to tell if a subroutine will perform the desired function? Are the calling conditions and parameters clearly documented? Are all of the side effects clearly documented?

Reviewing your own code from an outside viewpoint helps to identify problems in these areas. A program that meets these standards is not only easy to maintain but is likely nearly free of bugs more serious than one-line slip-ups.

5. Structure

Programs should be organized in a tree, or multi-level pattern. The layout of the various procedures and structures of a program should reflect its structure. A typical program would be laid out as follows:

DOCUMENTATION HEADER

Title

Program Title
Author
Copyright
Date

Description

A general description of the programs use and capabilities.

Use

A brief description of the input arguments, command line, or whatever.

Special Information

Any information of importance that does not fit above. For example, a discussion of the algorithms used if they are externally visible or affect the proper use of the program.

Modification History

Documentation of changes made, by whom, when and why.

GLOBAL ITEMS

Global Declarations

Major declaration of symbols and '#defines' which effect the program as a whole, or which are used to configure the program. Declarations of flags and values used in specific structures are normally defined immediately below those structures.

Global Data

Global data structures are discussed. Pay particularly close attention to see that global values are clearly documented as to the times and conditions in which they are valid. It is preferable that all applicable variables be valid at all times. Fields that have defined values should be followed by those definitions.

MAIN PROGRAM

The official main program should be short and clear. This is where people will first come when they're looking for something. Its structure should be simple and clear:

Decode switches, flags, environment
Decide what to do
Prepare to do it
Do it
Clean up

Usually most or all of these functions are independent subroutines.

SUBROUTINES

Top Level Subroutines

These are the primary subroutines of the program. They either perform or organize the bulk of the work. For example, a compiler might have top-level subroutines of

Init - Initialize Memory and Tables
Pass1 - Syntax and Semantics
Pass2 - Code Generation
Summ - Close files, print results

The routines called in the main program are usually included in the top-level subroutines, even though they may be relatively simple and unimportant. This allows them to be found easily when the code is being scanned.

Top Subroutine Helpers

Dedicated subroutines used by those in the top level go here. These are normally 'lieutenant' subroutines, performing specific tasks for a one or more top-level routines.

Low-Level Subroutines

These routines are usually relatively small, short and perform a general utility service. They are often called from several places and perform services such as symbol table management, and data copies. They are the fundamental items of the program abstraction approach. They are usually arranged alphabetically since they are called by (perhaps) several routines at all levels.

6. Naming Conventions

Variable, subroutine, and structure names represent an important opportunity to document and describe a routine. Names should be carefully chosen to avoid an "impedance mismatch" between the program and its reader. Names that are too long distract the reader from the structure of the program. Likewise, short names are too cryptic. A variable name such as 'i' is appropriate for a local scratch value, perhaps used in a two or three-line loop. 'i' should be used for signed integers, 'u' for unsigned, 'f' for a floating scratch, etc. Variables with a longer range of significance should have a longer name to remind the reader of its purpose.

Note that by using a one or two-character variable name you're effectively documenting that this variable has a very local range. When a future programmer adds code to the routine he should be able to assume that if the immediately surrounding statements are not using the variable then it is free for his own short-term (code lines, not execution time) use.

Variable names should be chosen to minimize confusion. For example, long names which vary in the last character position are unsafe. Variable names that mix similar characters such as letter o and digit 0 or letter l and digit 1 are also unsafe. If possible, choose names that are "pronounceable" (xpos, not xpstn). When abbreviating, always keep first letters and be consistent (xpos and ypos, not xpos and ypstn). If an extensive naming convention such as 'Hungarian' is used, then a glossary of root terms should be generated and inserted into the headers of all files that compose the program.

In general, the form of a name should indicate its type. Local variables should be all lower case. Global variables should be lower case with the first letter upper case. '#define' variables should be all upper case. Bit, flag, and structure items should contain an underscore and be clearly related to their parent item. For example:

```
unsigned int Sysmod;      /* system mode control */
#define SM_USR  0x001     /* user mode */
#define SM_ROM  0x002     /* ROM enable */
#define SM_WRM  0x100     /* warm start */
```

Note that all variables and procedures should have their type explicitly declared, even if that type is 'int'. A procedure of undeclared type is assumed to return no value.

7. Comments

Comments are intended to rapidly communicate information from the program developer to people that are reading the program for modification or algorithm extraction purposes. To properly perform this task, comments should contain information not readily apparent from the code, or should provide a road map to the program. All programs and subroutines should contain the header description block described in this document. High-level language programs should contain comments that describe the purpose of blocks of code. If a statement is particularly tricky, a comment should be added to describe its purpose. Assembly language routines should contain sufficient comments to allow a reader to easily follow the instruction flow. Please note that for comments to rapidly transfer information, several conditions must be met:

- The comments must accurately describe the program. Misleading, cute, or incorrect comments are worse than none at all.
- The amount of commenting must be appropriate. There is no magic formula to determine the correct amount. Too few comments don't convey the information; too many or too wordy comments obscure the information.
- Comments should not contain non-standard abbreviations. The purpose of comments is to communicate information. Non-standard, essentially random, abbreviations impede this communication by forcing the reader to stop and determine the real meaning of the abbreviation. For example, ptr is the accepted abbreviation of pointer and using pnt, pntr, pointr impede communication. The excessive use of random abbreviations leads to statements like: Cmnts shld nt contn nstd abbr.
- Comments should describe the *why* more than the *what*. Avoid comments which describe the evident action ("add BX to AX"); use comments which describe unobvious action or the purpose ("add subtotal to total").

8. Coding Conventions

The coding conventions described in this section are often arbitrary. The critical purpose is to maximize the ease of viewing and to provide visual uniformity to our code. Note that this implies that if you are modifying a piece of existing code you should use the existing conventions rather than the ones described here. This particularly true if the mixture of two or more styles will complicate the reading of the program. New routines and major new code segments should follow these guidelines.

8.1. General

Whether coding in 'C' or assembly, certain conventions should be followed:

- Individual subroutines should be kept, whenever possible, to one or two pages.
- Within a subroutine, code should be grouped into sections with each section set apart with a banner. For example, in Figure #1, on the following page, the sections "scan forward..." and "have a difference..." are made visually prominent with a comment banner surrounded by white-space.
- White space should follow every unconditional transfer. This emphasizes the break in control structure.
- Comments should be helpful, not superfluous. Do not describe machine/language operations, for example:

```
lcnt &= ~1;     /* clear low order bit */
```

When the action is obvious, omit the comment or use the opportunity to explain a motivation, or longer-term goal. Merge comments when effective:

```
chcnt &= ~1;
wdcnt &= ~1;        /* make counts even */
lncnt &= ~1;
```

- Comments should be designed to optimize the transfer of information to the reader. Too few are cryptic, too many are obscuring. A common mistake is to test for some condition and have the comment refer to the truth or falsity of the inverse of the condition. The code and comment flows should parallel each other.

```
/***
 *                SUBROUTINE HEADER
 *
        .
        .
        .
/*  scan forward until we see a difference
 *  between old and new                    */
    for (pos=0; ; ) {                /* we exit loop via *return* */
        for (; *old++ = = *new++; pos++)
            if (pos = = SCRNSIZ) {
                fflush (stdout);          /* at screen end - done */
                return;
            }
/*      have a difference. Decide if we can write over
 *      to it, or if we should issue a new cursor sequence */
        if (((pos - cursor) < 4) && (pos >= cursor)) {
            for (; cursor != pos; )              /* write over to the spot */
                putchar (newscr[cursor++]);
        } else [
            .
            .
            .
    }
```

Figure 1.

8.2. Documentation Headers

As stated above, the main program and every subroutine should be proceeded by a documentation header. The basic form of this header is:

```
/***    name - very brief description
 *
 *      <description>
 *
 *      name    (parm1, parm2, parm3,..., parmn)
 *
 *      ENTRY   parm1 - description
 *              parm2 - description
 *      EXIT    parm3 - description
 *              .
 *              parmn - description
 *
 *      <discussion of internals, one or more paragraphs>
 *
 *      WARNING:  <discussion of limitations, gotchas, etc.>
 *
 *      EFFECTS:  <'none', or discussion of global effects>
 *
 */
```

The initial banner is preceded by 4 blank lines. This white space together with the asterisk pattern serves to delineate the subroutine and make it stand out to the eye. The name should be reasonably short and mnemonic; frequently used routine and/or function names should be short enough to scan easily. The first line in the subroutine header just gives the name and a very brief description. For example,

```
/***    move - move memory
/***    symtab - lookup/insert in symbol table
```

The next header paragraph describes the routine's function in more detail. This is still basically a summary; it describes what the routine does - but not usually how it does it. This paragraph should be concise and readable. Any necessary blow-by-blow narrative can come later. Of course, in simple enough cases, this paragraph might tell all:

```
/***    move - move memory
 *
 *      Move moves an arbitrary block of bytes in memory.
 *      The moving is done so that overlapping fields
 *      will not ripple.
 *
```

The next section of the header describes the calling sequence. The arguments are each briefly discussed. The entry arguments are described in contiguous lines followed by the exit arguments. Usually the argument name and a one-line discussion are adequate:

```
*               from - source address
*               to   - destination address
```

Sometimes 'see below' is added to indicate further discussion. In a few cases it might be necessary to postpone the calling sequence discussion until further detail paragraphs have been presented.

The subroutine header must describe not only the non-local effects it personally generates but also those that its callees generate. A subroutine header should thus describe all the possible consequences that a call to that subroutine might have. Once again, in the interests of simplifying a subroutine's description, we would normally say "Modifies ralph" rather than "Modifies ralph if this, and not that ...." If this causes extra code for a few calling routines to save the value of "ralph" when we 'know' that "ralph" will not be modified, then so be it. Normally, we should be willing to sacrifice a few bytes or microseconds for a simpler structure. In those cases where performance is critical we should still make the 'circumstances' list as simple as possible.

Following the summary and calling sequence are the detail paragraphs. These paragraphs discuss pertinent information of a more global nature that are of concern to the caller, or that might not be directly obvious from the code. Any unusual details of the code should be discussed. For example:

```
*       This algorithm is not very fast, but it is only
*       called once per compilation.....
```

or

```
*       This routine contains special code to use
*       tabs instead of blanks, if the variable
*               Usetabs
*       is true.
```

At the end of the procedure header come two important sections, **WARNINGS** and **EFFECTS**. These are placed at the end so that they make sense if they refer to implementation details, and so that they also 'stand out' to the eye. The **WARNINGS** section details any restrictions or limitations on the implementation that might cause grief for a maintainer. The **EFFECTS** section details any global effects that the procedure might have. Note that global effects are not restricted to just modifying global cells; any action not strictly local to the procedure should be mentioned here. For example:

```
*       EFFECTS:  may allocate memory
*                 may repack tables (via 'packtab')
```

As we mentioned earlier, any globally visible result - such as a kernel routine modifying the CPU priority (via spl) - is an effect. In fact since routines should normally restore the current priority changing the priority is serious enough that it should probably appear in the **WARNINGS** section, as well.

8.3. C Language Conventions

As stated above, the general form of a 'C' program is:

Documentation header
Global definitions
Main program
Top level subroutines
Utility subroutines

Before the main routine and each subroutine is a documentation header as defined above. The first line of the header should begin with /*** in column 1 and all subsequent lines should contain an asterisk in column 2. The comment block terminator */ should be on a separate line in column 2. Even if the purpose and inputs of a subroutine are 'totally obvious', the header should be included to make the subroutine stand out and easier to understand. The top level and utility subroutines should be alphabetized within each group. This makes it easier to locate subroutines when reading or debugging the program.

After the documentation header comes the include statements required by the program. The included files from the standard library are specified first followed by those unique to the program. Each of these sets is preceded by comment lines and the comment lines are preceded and followed by white space. The comment lines for the non-standard #includes should give some indication of the what is being included. This assists an unfamiliar reader in determining whether or not to locate and study the contents of the #include file. It should also be noted that #include files should not contain initialization of any variables. Otherwise, each use of the include file will cause the variable to be reinitialized. Also, include files should not be nested.

Next comes the local declarations. The general order is first the **extern** statements, then **register**, then **auto** followed by **static**. These declarations are sorted alphabetically by type within each class and then alphabetically by name within each type. Comma separated lists may be used. If a variable is initialized at declaration time it should be on a separate line. The types of all variables must be specified on the declaration even if of type int. In particular, the formal parameters to a function must be declared and not allowed to default to int. Please note that we have just described the *general* order. Variable items which are functionally coupled should be declared together - the ordering guidelines should not be allowed to interfere with understandability.

Although the 'C' compiler will accept input lines of any length, it is good practice for programs to fit into lines of 80 characters or less. Creation and modification of 'C' programs is performed using interactive terminals, and the wrapping of long lines obscures the structure of the program.

Parentheses should be used whenever there is a possibility of ambiguity. Note that the precedence of operations in 'C' can cause surprises. In particular, & and | mix poorly with the relational operators and the assigning operators are weaker than && and ||. As an example:

```
a & 030 != 030
```

will be interpreted by the compiler as:

```
a & (030 != 030)
```

which is probably not what the programmer intends. The shift operators << and >> generally cause surprises unless properly parenthesized.

Statements are formatted to emphasize control structure according to the patterns shown in *Figure 2*. Note the use of opening and closing curly-brackets - it is important that we remain consistent in their use so that we can read each other's code easily. Indentation is normally done by tabs (8 characters) but this can be reduced to 4 when the need arises. As always, our goal is to employ the physical layout of the statements - including white space - as a way of illustrating the design and flow of the code.

Within a statement there should be no blank lines, tabs or multiple blanks. Each keyword should be proceeded by white space (tabs or a single blank) and followed by a single blank. Binary operators should be preceded and followed by one blank character. Unary and addressing operators should not be separated from their operands by white space. Calls to functions should be written as the function name immediately followed by the opening parentheses. Following the form in all cases improves program readability.

If an expression is too long to fit on a line it should be continued on the next line indented further than the start of the statement but less than one full tab stop. For example, the continuation of an if statement would line up one character to the right of the opening parentheses of the condition:

```
if (v1 == cond1 && v2 == cond2 &&
    v3 == cond3) {
```

If a statement is continued, it is a good rule to have a trailing operator on the preceding line or to continue within a parentheses group. These displaced fragments are more likely to cause compiler diagnostics if the program is later incorrectly modified.

8.4. Assembly Language Conventions

8.4.1. Form

The coding standards for assembly language programs are essentially those for *C*, or any other language we use. The initial expository comments, procedure headers, section banners and so forth should be present in assembly language programs, as well. Further, each procedure header should have a **USES** section which describes register usage - since each register is a special kind of global variable, their modification must be reported. See *Figure 3* at the end of the document for a sample of assembly language header.

Note the convention used to describe register contents:

```
(r1)      Contents of r1

((r1))    Contents of location pointed to by (r1)

((sp)+2)  Second item on the stack
```

The stack frame itself is an important global item and must be discussed when used in a non-trivial fashion.

The indentation conventions discussed for high-level languages should not be used for assembler - it becomes too difficult to scan the code when the columnar layout is broken up. Careful use of white space substitutes for statement indentation.

8.4.2. Lexical Layout

Assembly language programs are, in a lexical sense, larger and more complex than those written in a high level language. For example, they must manage and describe the use of not just local and global variables, but also the register set which is used as a variable cache.

This higher level of complexity means that the assembly language programmer must take extraordinary care in using the lexical layout and structure of his program as a tool to document the structure and functioning of his subroutines. A well thought-out and uniform standard for the use of white space, blank lines, comment sub-banners, etc., makes the difference between a "puzzle program" that forces the reader to "play assembler" to read it and a program that virtually explains itself upon a casual scan.

8.4.2.1. Blanks after unconditional jumps

One of the simplest and most powerful lexical techniques is the separation, by blank line(s), of the various "paragraphs" of code. Every unconditional branch should be followed by a blank line; the white space makes clear the termination of the instruction sequence and marks the beginning of a new one.

8.4.2.2. Comment Sub-banners

A typical subroutine has one or more "watershed" tests in it wherein it chooses between two or more major courses of action. For example, the flow in a symbol table manager probably divides based upon whether or not the symbol is already in the table. These major decision points should be delineated by a comment sub-banner of the form:

```
                <2 blank lines>
;*      Symbol already in the table.
;
;       <description of action to be taken>
;
;       (DS:DI) = ptr to Symtab. record
;       <other register contents and required items discussed>
                <blank line>
```

and elsewhere:

```
                <2 blank lines>
;*      Symbol not in the table.
;
;       <description of action to be taken>
;
;       (ES:SI) = ptr to name string
;       <other contents and requirements discussed>
                <blank line>
```

More commonly a test is made of somewhat less impact on the execution of the subroutine. An example might be the discovery that a buffer has become full and must be emptied.

```
                <1 blank line>
;       Buffer is full.  Empty it and reset pointers
                <blank line>
```

These lessor comment-subbanners often omit a discussion of register contents; omission is appropriate when the variable and register contents have been well described nearby or have no bearing on the section of code being conditionaled. If the situation is more complex or there are no other "nearby" discussions of register and variable contents then they should be discussed here.

8.4.3. Control flow

Control flow can be a major stumbling block to the design and understanding of assembly code; control flow also represents a major opportunity for explanation via lexical conventions.

8.4.3.1. No jumps back up

Although not strictly a lexical consideration, its important that code only contain jumps "forward", to later labels in the program. The use of this rule simplifys the control structure of a program and greatly reduces the confusion about register contents, a prime source of bugs.

When one wishes to modify a section of code a crucial question is always, "what is in the registers" and "what registers may I modify?" Subroutines which jump every which way make these questions extremely difficult to answer, since the register contents depend upon the jumps which have been taken to the particular code location. If only forward jumps are used its easy to scan backwards through the code and see the register contents at the time of the first jump, then to scan forwards to see which of those contents are still valid in the case that execution "fell through" to the label.

Naturally, there is an obvious exception to this rule: program loops. These are discussed below. Another common excuse for an exception is to share a common code tail, especially a "subroutine exit" code tail. These should indeed be shared, but should be placed at the last location they are needed and having earlier references make forward jumps to the tails.

8.4.3.2. Program Loops

All program loops (except for trivial "4-liners") should start with a comment subbanner describing the purpose of the loop and describing the register contents at the top of the loop. This is very important as it deals with the major problem of a label which can be reached in more than one way (start-loop and continue-loop): what registers contain what values, and which may be used in the loop body.

8.4.4. Miscellaneous

There are a few additional points of assembly language documentation style which are a great boon to understandable code:

- register content description

  Comments which document the loading of a register should take the form:

  (regname) = description

  where *description* should be as full as possible while still being short enough to scan easily. For example, use

```
MOV     BX,SI           ; (BX) = symtab entry pointer
```

instead of

```
MOV     BX,SI           ; save pointer
```

Comments of this form help the reader determine the current contents of the registers because its easy to scan backwards and pick out the last setting of the register.

- Comments on jumps

  The two key points when documenting a conditional jump are to describe the condition being tested and the sense of test: are we jumping if the condition is true or false? The comment on a jump instruction should always describe the situation if the jump is *taken*; the fall-through situation may be obvious, it may be described in the comment on the first fall-through instruction, or it may be described in a fall-through comment sub-banner. For example, the statements

```
CMP     AX,BX           ; see if table empty
JE      lab1            ;  yes!
```

should be coded as

```
CMP     AX,BX
JE      lab1            ; table is empty
```

8.4.5. Label Names

When we learn to program we are all taught the importance of mnemonic program labels. However, in large assembly language programs, mnemonic labels such as "yes", "loop" or "found" can cause more problems than they solve. First, labels which are mnemonic to the program implementor may not have the same value to somebody else (including the implementor) several months or years later. Second, with the normal restriction of eight character labels it is difficult to create enough labels with mnemonic value because of the large number of labels required. Third, it is difficult to locate program labels if they are created essentially at random. This makes reading and debugging of assembly language programs even more difficult.

For these reasons, it is recommended that program labels be synthesized as

<name><number>

or

<name><alpha>

where <name>, <number> and <alpha> are derived as follows:

<name> is a mnemonic abbreviation of the subroutine purpose such as "lte" for "locate table entry" or "crv" for "compute radix value". <name> is normally three characters long.

<number> is a sequential ascending number assigned to branch labels within the subroutine. For example, if <name> is crv, then the first branch label is "crv1", the second is "crv2" and the tenth is "crv10." If a program is later modified and additional branch labels are inserted, they have the format

<name><number.insert>

where number is the original label number above the inserted label, "." is some valid non-alphanumeric character and <insert> is a sequential ascending counter. For example, if three labels are

inserted after "crv2", they would be called "crv2.1", "crv2.2" and "crv2.3". Two special forms exist for subroutine branch labels: the entry point of a subroutine has the form <name> and a common exit point has the form <name>x.

<alpha> is a sequential ascending alphabetic string assigned to labels used for constants or variables local to the subroutine. Using the above example, the local variables for "crv" would be named "crva", "crvb",... In the case of self-modifying code at a branch, there would be two labels, one of the form <name><number> to indicate a branch and one of the form <name><alpha> to indicate a variable that is stored into.

A naming convention of this type has the following advantages:

- The generation of labels is automated.
- The label type is classified by the suffix of <number> or <alpha> and inadvertent stores into code or branches to variables is minimized.
- Since subroutine labels are assigned sequentially it is easy to locate labels when reading and debugging code: if the label is larger than the closest label, look further down the page; if the label is smaller, look up the page.
- Since all of the labels for a subroutine have a similar format, it is easier to spot code that is out of place. It is also easier to identify the transition from one subroutine to another.
- Since all of the internal labels of a subroutine are of the format <name><string> and the only valid entry point has the format <name>, it is easy to detect branches into the middle of a subroutine body.

If the assembler allows local labels such as 1$, 2$, etc., they may be used. If local labels are used they should still be in ascending numeric order and unique to the subroutine. Initially, the local labels can be incremented by some constant value to allow the insertion of new labels during debugging and maintenance.

9. References

The following books are recommended for further reading about programmers and programming style.

The Elements of Programming Style, 2nd ed., Kernighan and Plauger, McGraw-Hill Book Company, 1978

The Mythical Man-month, Frederick P. Brooks, Jr., Addison-Wesley Publishing Co., 1975

Software Reliability, Principles and Practices, Glenford J. Myers, John Wiley & Sons, 197x

The Psychology of Computer Programming, Gerald M. Weinberg, Van Nostrand Reinhold Co., 1971